Agnostic Middleware for VANETs: Specification, Implementation and Testing

Fábio Gonçalves, Bruno Ribeiro, Oscar Gama, Maria João Nicolau, Bruno Dias, António Costa, Alexandre Santos and Joaquim Macedo
Centre Algoritmi, Univ. Minho, Portugal

Keywords: VANETs, Middleware, Agnosticism, Protocol, API.

Abstract: Vehicular Ad hoc Networks (VANETs) are the basic support for Intelligent Transportation Systems (ITS), providing a framework for its multiple entities to communicate. The communications and services provided to the road entities are generally implemented by means of an On-Board Unit (OBU) and Road Side Unit (RSU), sharing a rather similar hardware and software architectures. These devices need to support multiple communications types and provide access to the needed vehicle data to the applications. Most of the existing solutions demand that the application developers have development control and knowledge about the OBU internals. Thus, most applications are developed by OBU makers and implemented directly on it. However, solutions based on middleware agnosticism separate the OBU internals and software development, facilitating the application development by software makers or researchers. This paper proposes an Application Programming Interface (API) and protocol that can easily be used to access the vehicle internals and services through an agnostic architecture without any knowledge on how they are implemented.

1 INTRODUCTION

Intelligent Transportation Systems (ITS) is a set of applications and services that aim to facilitate transportation and make roads safer. It allows avoiding road obstacles and traffic redirection.

Vehicular Ad hoc Networks (VANETs) are the underlying communication network that enables the several ITS nodes to communicate. The nodes can be divided into infrastructural and road entities. The first are usually entities with no movement located in the infrastructure, and the latter the road vehicles. Depending on their physical location, the device that enables vehicular communication to implement the multiple services has different characteristics and designations; if it is located in the infrastructure Road Side Unit (RSU), if on the road vehicles On-Board Unit (OBU).

In the traditional approach, and following the existing architectures (ETSI ITS-G5, Communications Access for Land Mobile (CALM), etc.), each manufacturer develops its own applications and deploys them directly in the OBU, transforming them into a black box with a set of services and applications. Thus, making it impossible for the development of third-party applications and complicating the interoperability with OBUs from other manufacturers.

Authors in Dias et al. and Sousa et al. (Dias et al., 2018; Sousa et al., 2017) propose a new approach on communications architectures for ITS by adapting existing communications models defined, by the most important standardization institutions, into a more specific and modular architecture. It models the OBU like a black box where all the lower-level services are implemented. These services share the same interface technologies and implemented functionalities, covering the same functionalities as referred by European Telecommunications Standards Institute (ETSI) ITS. Its main goal is to make it easily adopted by the manufacturers and research development and integration of independent application-level software.

This agnostic architecture provides a separation between the OBU internals and high-level application development through the ITS-Local Communication Interface (ITS-LCI) (Dias et al., 2018; Sousa et al., 2017) to interact using standard communication protocols and access technologies implemented on the three service modules: Communication Services Module, Information Services Module, Function Services Module.

The ITS-LCI should use a large bandwidth, low-

Gonçalves, F., Ribeiro, B., Gama, O., Nicolau, M., Dias, B., Costa, A., Santos, A. and Macedo, J.
Agnostic Middleware for VANETs: Specification, Implementation and Testing.
DOI: 10.5220/0011315100003286
In *Proceedings of the 19th International Conference on Wireless Networks and Mobile Systems (WINSYS 2022)*, pages 84-92
ISBN: 978-989-758-592-0; ISSN: 2184-948X

cost medium wired technology like Gigabit Ethernet, support a network stack like User Datagram Protocol (UDP) over Internet Protocol (IP), and an application management protocol like Simple Network Management Protocol (SNMP) (Harrington et al., 2002).

This work presents a well-defined protocol and Application Programming Interface (API) for access to the OBU's lower layer services through the ITS-LCI. Thus, allowing easy development of third-party applications without the specialized knowledge of the OBU's internals. The proposed protocol uses UDP and is encoded using the standard widely known Abstract Syntax Notation One (ASN.1) format, facilitating an easy and quick development and deployment in any system.

The remainder of the paper is organized as follows. Section 2 discusses the related work, presenting other middleware architectures in the literature. Then, in Section 3, the agnostic architecture that serves as the basis for the development of this work is described. Section 4 presents the API designed to access the OBU internals through the agnostic architecture. Section 5 evaluates the designed protocol, and, finally, the conclusions are presented in Section 6.

2 RELATED WORK

VANETs are complex and heterogeneous networks, and in the future, vehicles may use multiple types of wireless communications. Thus, the implementation of applications and technologies should be independent of the underlying medium access communication framework. Additionally, the same concept should be applied to the transport and network level. This can be achieved by applying a concept known as an agnostic middleware communication layer. It can provide independent information management for multiple data sources, allowing the communication over heterogeneous interfaces to be transparently supported over different stacks.

CALM is a communication architecture defined by the ISO Technical Committee 205 - Working Group 16 (TC204 WG16). It is designed to allow interoperability between Cooperative Intelligent Transportation Systems (C-ITS) stations, and it abstracts applications and services from the underlying communication layers (Willke et al., 2009). The CALM architecture defines a set of standard specifications (Dias et al., 2018; Sousa et al., 2017), including ITS Station Management, Communications Security, Facilities, Station Networking, Transport layer protocols, Communication Interfaces and Services, Interfacing Technologies and ITS Station, and Distributed Implementations ITS Stations, Interfacing ITS Station to existing communication networks. . However, it does not present detailed solutions for simultaneous usage of the different Vehicle to Anything (V2X) communication technologies by multiple applications in the same ITS station.

ETSI ITS-G5 is defined by the standard ISO 21217:2014 (WG16, 2014), which describes the ITS station reference architecture. It consists of six parts: Applications, Management, Facilities, Networking and Transport, Access and Security. However, the standard does not specify whether a particular element should be implemented. It depends on the specific communication requirements. The most relevant middleware support solution for application agnosticism architecture on CALM was introduced by the facilities layer (Some more recent alternatives were proposed (Nour et al., 2011) (Costa, 2013) (Silva et al., 2013) (Jawhar et al., 2013), but they can be seen as forms of instantiation of the generic ETSI proposal). However, the implementation of such middleware might be very complex, needing knowledge of many details of the lower layers. Thus, causing software makers to lose their ability to choose the development paradigm for their applications.

3 AGNOSTIC ARCHITECTURE

The Agnostic and Modular Architecture for the Development or Cooperative ITS Applications is an architecture adapted from the modern ETSI and ISO to be deployed on ITS and overcome the current architecture standards shortcomings. Its goal is to enable middleware agnosticism, facilitating the development of cooperative ITS applications and services that use different communication technologies and network/transport stacks to use the communication medium transparently. It separates the OBU internals and the application development, enabling an easier integration of applications developed by third parties. Furthermore, the separation between user applications and OBU internals more easily fulfills the strict safety and security requirements of the automotive industry's strongly regulated manufacturing and deployment processes.

The agnostic ITS architecture supports all types of vehicular applications using any programming paradigm and taking full potential of a multi-medium access capable OBU.

For this approach, the OBU should provide a group of services, implementing all common features needed for access to the vehicle's internal data sources, V2X communications, and lower-level automotive functions. The interface to these services is identified as ITS-LCI. It should be available through technologies that are widely used and easily supported by all automotive manufacturers. Software applications implemented in the ITS station outside the OBU would use the technologies defined for this interface for access to the OBU internals.

As shown in (Dias et al., 2018; Sousa et al., 2017), the OBU implements three modules, Communication Services Module, Information Services Module, and Function Services Module, that are accessible through the ITS-LCI.

Communication Services Module - Must permit sharing of all the available OBU's medium access technologies, implement adequate multi-homing algorithms, vertical hand-off, and related functions for medium-access addressing, monitoring parameterization, and security. Also, when available, implement antenna configuration as no direct access to communication functions is possible from external services.

This module requires a bidirectional adaptation to allow forwarding Protocol Data Units (PDUs) arriving in the communication block/channels through the ITS-LCI to the adequate application stack at the network level, depending on their type, while implementing efficient scheduling strategies.

Information Services Module - Must permit access and manipulation of data generated by all sensors, actuators, and other devices indirectly or directly connected to the OBU at rates and precision adequate for the proper use of applications, some in almost real-time. Additionally, parameters such as sampling rate and security access should be configurable, preferably through configuration functions, but direct manipulation, as long as adequate security mechanisms are supported.

Function Services Module - Must permit access to lower-level functions. These are atomic operational procedures implemented by the manufacturers due to security, safety, performance, and liability issues. These procedures can then be used to implement all types of external applications with a higher-level functionality.

4 AGNOSTIC MIDDLEWARE API

The ITS-LCI represents a standard bidirectional communication technology that all parties need to implement to be able to use the functions provided by the three previously mentioned services.

This section describes the protocol and API developed to allow any third-party application to easily access the OBU internals without any knowledge of how they are implemented.

The architecture (Figure 1) assumes two entities: the communication server (to be deployed as the OBU board) and the client application (to be deployed as a host system on the same local area network).

Each application can implement its own network protocol, independently of the communication interface. The developed API provides a standard way to access the services previously described.

Communications between applications and the OBU are made over the widely known UDP, a very simple and light internet protocol. It has no handshake or confirmations, but the communications between all the parties happen inside the vehicle, in close proximity, and over a high bandwidth physical technology. Thus, the probability of errors or packet loss is low. This protocol benefits, however, from its low overhead.

The OBU can be accessed in two different ports, depending on the operation type, as shown in Figure 1. In port 9011 is implemented a data exchange service to be used by the client application. The sent or received data is either typed (Context Awareness Message (CAM), Decentralized Environmental Notification Message (DENM), Basic Safety Message (BSM), Wave Service Advertisement (WSA), etc.) or opaque.

Configuration of the server is attained by setting configuration communication parameters through primitives addressed to the configuration server UDP port 9012. This port can also be used to check the server configuration.

The port marked as XXXX and YYYY are randomly attributed when opening the socket. The XXXX port is used by the client application to receive the response_requests. The YYYY port is merely indicative. It is a result of opening a socket for the client application on the C_PORT.

The API's implementation uses different UDP ports for the different operations, thus, isolating the different operations and simplifying the API's implementation.

The C_PORT is used by the client application. It is defined by the application at the moment of creation. It allows a device to have multiple applications running and using the OBU's services.

In the proposed protocol, the OBU is a communication server that implements a data exchange communication service to be used by any client ap-

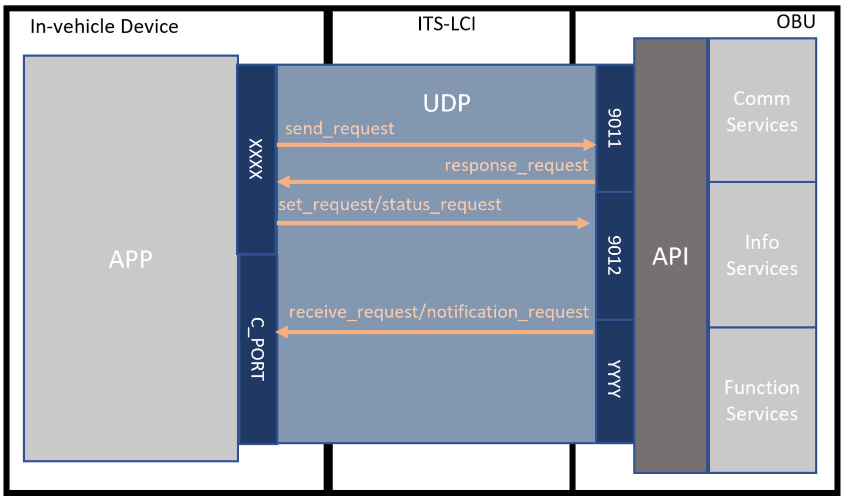

Figure 1: ITS-LCI communication architecture.

plication. The services are asynchronous and non-confirmed over UDP.

4.1 Message Format

This section presents the different types of messages supported by the current protocol version, their payloads, and functionalities. Table 1 presents the multiple configuration parameters that can be altered in the OBU. It shows the identification code to be used, the value type, and the parameter's name and description. These can be used in the set_request, status_request, and notification_request messages.

Table 2 shows the multiple error codes defined and their description. These errors are only used by response_request and notification_request messages.

The message format to be used for the communication between applications and the OBU is as follow:

- **[Version] [Message ID] [Payload]**

The field **Version** is an Integer that indicates the current version of the protocol *0x01* is the first version. **Message ID** represents the message identifier, which is a 32-bit random value that univocally identifies a particular request primitive through a reasonable time period. The last field, **Payload**, is the payload of the message. One of the 6 different payloads is permitted:

- *send_request*: Used for requesting the server to send the included data using the IEEE 802.11p protocol; the request should be sent to the data communication server UDP port; the server shall respond with a *response_request* to the origin IPv4 + UDP port of the client application. This payload type also has a **Data** field.
  - **[Data]**: Data can be untyped (Opaque) or one of the following types: CAM, DENM, BSM, WSA. When this field includes a typed data, it does not refer to an exact syntax of the message PDU of that standard; instead, it only specifies data values that can be obtained from one or more original standard message PDUs. It should contain at least one byte of data and not more than 1024.
- *receive_request*: This primitive is for requesting the client application to receive the included data; the request should be sent to the pre-defined client application UDP port; the client application shall not respond to this primitive. This payload type also has a **Data** field.
  - **[Data]** Same as the one for *send_request*.
- *set_request*: Primitive for requesting a configuration operation in the communication server; the request should be sent to the configuration server UDP port; the server shall respond with a *response_request* to the origin IPv4 + UDP port of the client application. Note: when making a pure abstract message specification, each tuple can be replaced by an abstract parameter field combining identification and value. This payload also includes a list of parameters to be configured. The list format is **[Parameter ID][Parameter Value]**.
  - **[Parameter ID]**: please refer to Table 1; please note that some configuration parameters are dependent on other higher-level parameters; for example, the values used for 802.11p channel numbers are dependent on the 802.11p regulation domain; also there would be one 802.11p channel number to be configured for each available medium antenna/port.: please refer to Ta-

ble 1; please note that some configuration parameters are dependent on other higher-level parameters; for example, the values used for 802.11p channel numbers are dependent on the 802.11p regulation domain; also there would be one 802.11p channel number to be configured for each available medium antenna/port.

  – **[Parameter Value]**: the size and meaning of the parameters depend on their ID; when the parameter ID refers to a parameter type that can have a variable size, the value encodings should include information about the parameter size and, optionally, other relevant encoding/decoding information.

- *status_request*: Primitive for requesting information about the communication server's configuration parameters; the request should be sent to the configuration server UDP port; the server shall respond with a *response_request* to the origin IPv4 + UDP port of the client application. This payload also includes a list with the IDs of the parameters to be consulted.
  – **[Parameter ID]**: please refer to Table 1; when there is the possibility of having multiple parameters with the same ID (like, for example, all parameters related to each medium antenna/port), the associated *notification_request* should only include the values of the parameters related to the set value of the respective higher-level parameter.
- *notification_request*: Primitive for requesting the client application to receive the included notification information; this information is directly related to previous send, set or status requests issued by the client application and that can be identified by the Message ID; when the Message ID refers to a previous *send_request*, than it permits the communication server to inform the client application of the success or failure of the data send operation (is this case, the Error Value indicates the code of the results of the operation); when the Message ID refers to a previous *set_request*, than it permits the communication server to inform the client application of the success or failure of the attempted configuration operation (is this case, the Error Value indicates the code of the results of the operation and the included list of parameters indicate the values after the attempted configuration operation); when the Message ID refers to a previous *status_request*, than it permits the communication server to inform the client application of the success or failure of the attempted operation (is this case, the *Error* value indicates the code of the results of the operation and the included list of parameters values indicate the current status of the configuration parameters); the request should be sent to the pre-defined client application UDP port; the client application shall not respond to this primitive. Note: when making a pure abstract message specification, each parameter tuple can be replaced by an abstract parameter field combining identification and value. The *notification_request* primitive includes also an error value and a list of parameters.
  – **[Error Value]**: the error code associated with the request primitive that the notification relates to; please refer to Table 2.
  – **[Parameter ID]**;
  – **[Parameter Value]**;
- *response_request*: Primitive for requesting the client application to receive the included response information; this information is directly related to the previous send, set, or status requests issued by the client application and that can be identified by the Message ID; the server shall address this primitive to the origin IPv4 + UDP port of the client application. This payload type also includes an **Error Value**.
  – **[Error Value]**;

Data to be sent by the communication server should be addressed to its data server UDP port 9011 using a send_request primitive. All information to be received by the client application should be addressed to a pre-configured client application UDP port (except for the *response_request* primitive). The *receive_request* primitive should be used by the server to send data to the client application on port *C_PORT*, while the *notification_request* should be used to send information about results on client requests to the server.

The communication server also implements a configuration communication service to be used by the application client. The service is asynchronous and non-confirmed. Configuration of the server is attained by setting configuration communication parameters through a *set_request* primitive addressed to the configuration server UDP port 9012. The client application can check for the server's configuration by issuing a *status_request* to the configuration server UDP port 9012.

The communication server should always respond with a *response_request* to any *send_request*, *set_request*, or *status_request*. The communication server should respond if the request was accepted or not (it does not imply any confirmation on the success

Table 1: Configuration Parameters.

ID	Value Type	Name	Obs
0x00	NA	All parameters	Indicates all parameters available
0x01	Unsigned Integer	Number of Antennas/Ports	Number of medium antennas/ports
0x02	Unsigned Integer	Active Antenna/Port	Antennas/Ports, when available should be numbered sequentially (0x01, 0x02, etc.); if no antenna is temporarily available then values should be 0x00.
0x03	Unsigned Integer	Active Channel Number	The active channel number (for example, 0xAE for channel 174) on the active antenna/port
0x04	Unsigned Integer	Active Channel Bandwith	The active channel bandwith/spacing, in MHz (for example, 0x0A for 10 MHz) on the active anetnna/port
0x05	Unsigned Integer	Active Channel Center Frequency Range	The active channel frequency range in MHz (for example, 0x16EE for 5870 GHz) on the active antenna/port
0x06	Unsigned Integer	Active Channel TX power	the active channel TX power, in 10^{-3} dBm (for example, 0x59D8 for 23 dBm) on the active antenna/port
0x07	Unsigned Integer	Active Channel Data Rate	The active channel data rate, in Kbits/s (for exampe, 0x1770 for 7Mbits/s)
0x08	Unsigned Integer	CAM Generation Rate	CAM generation rate in Hz (for example, 0x0A for 10Hz); if generation is 0x00 the CAM generation is off
0x09	Unsigned Integer	CAM Echoing Mode	If value is 0x00 CAM echoing to client application is off, otherwise it is on
0x0A	Unsigned Integer	DENM Generation Mode	If the value is 0x00 the generation is off, on otherwise
0x0B	Unsigned Integer	DENM Echoing mode	Same as CAM echoing mode
0x0C	String, 4 characters	Pre-defined IPv4 Address of the Client application	”0.0.0.0” or ”255.255.255.255” for broadcast on the IPv4 local network.
0x0D	Unsigned Integer	Pre-defined UDP address port of the client application (C_PORT)	If the value equals to 0x00 then no client application is set (for example 0,0x1F90 for port number 8080)
0x0E	Unsigned Integer	Server Request Time Out	The amount of milliseconds that the client should wait for the respective server's *response_request* before it considers it was not received, ignored or unattended (for example, 0x64 for 0,1 seconds)

or failure of the sending of the data through the configured communication technology or the success or failure of the setting request). The *response_request* should always be sent to the client IPv4 address and UDP port of the client application's original request, and it could be sent from any ephemeral UDP port on the server (implementing a concurrent UDP server), or from 9011 or 9012 UDP ports. After a period of time defined by **REQUEST_TIME_OUT** (please refer to 1), the client application can assume the request was not received by the communication server.

The client application never responds to any primitive received from the communication server, and all communications are insecure.

Table 2: Error Codes.

ID	Semantics
0x00	No error
0x01	Undefined Error
0x02	Service not available at that port
0x03	Unrecognizable message type
0x04	Maximum Payload Length surpassed
0x05	Unrecognizable message type
0x06	Unrecognizable data type
0x07	Unrecognizable Parameter ID
0x08	Mismatch on Parameter value
0x08	Unacceptable parameter value
0x0A	Unsupported parameter value
0x0B	Invalid number of parameters
0x0C	Unable to send data
0x0D	Not enough memory

Table 3: Protobuf and ASN.1 notation comparison.

Protobuf	message Student {
	  required int32 id = 1;
	  required string name = 2;
	  optional string email = 3;
	}

ASN.1	Student ::= SEQUENCE {
	  id INTEGER
	  name OCTET_STRING
	  email OCTET_STRING OPTIONAL
	}

4.2 Data Structure

The goal of this API is to provide a methodology for easy access to the OBU internals. Thus, the data should be structured using a widely known format that simplifies its adoption and interoperability.

Two different methods of describing the structured data were considered, Google's protobuf (Google, 2022) and ASN.1 (ITU-T, 2022). In Table 3 is shown a simple example of both methods.

Protobuf (Google, 2022) is a language-neutral, platform-neutral mechanism for serializing structured data. The data structure can be defined and then use a specially generated code to easily read and write this data to and from a variety of data streams. It is, comparatively, more recent, but it has gained some momentum in the last years.

ASN.1 (ITU-T, 2022) is a format for describing data transmitted by telecommunication protocols. It is independent of the language implementation and the physical representation of the data.

Ideally, the chosen description language should have tools that enable its translation directly to code, thus, simplifying the implementation in any system or language.

Google's protobuf is more recent, and there are far fewer tools, especially aimed at *ARM* processors. The one tool found presented some bugs, crashing the application each time the workload was increased.

ASN.1 has had more time to grow and evolve and has many more tools available that allow its conversion to multiple programming languages. So, it was the description format chosen for the development of the description of the messages. The data definitions are shown next.

```
LTS-LCI DEFINITIONS AUTOMATIC TAGS::=
BEGIN

MessageHeader ::= SEQUENCE {
version INTEGER,
messageID INTEGER
}

CAMp2aMessage ::= SEQUENCE {
stationid INTEGER,
timestamp INTEGER,
latitude INTEGER,
longitude INTEGER,
heading INTEGER,
speed INTEGER,
acceleration INTEGER,
yawrate INTEGER
}

Parameter ::= SEQUENCE {
numberofantennas INTEGER OPTIONAL,
activeantennaorport INTEGER OPTIONAL,
activechannelnumber INTEGER OPTIONAL,
activechannelbandwith INTEGER OPTIONAL,
activechannelcenterfrequency INTEGER OPTIONAL,
activechanneltxpower INTEGER OPTIONAL,
activechanneldatarate INTEGER OPTIONAL,
camgenerationrate INTEGER OPTIONAL,
demngenerationmode INTEGER OPTIONAL,
demnechoingmode INTEGER OPTIONAL,
clientappip OCTET STRING OPTIONAL,
clientappcport INTEGER OPTIONAL,
serverrequesttimeout INTEGER OPTIONAL
}

DataType ::= ENUMERATED {
opaque (0)
}

Error ::= ENUMERATED {
noerror (0),
undefinederror (1),
servicenotavailableatthatport (2),
unrecognizableversion (3),
unrecognizablemessagetype (4),
maximumpayloadlengthsurpassed (5),
unrecognizabledatatype (6),
unrecognizableparatervalue (7),
mismatchonparametervalue (8),
unaceptlableparametervalue (9),
unsportedparametervalue (10),
invalidnumberofparameters (11),
unabletosenddata (12),
notenoughmemory (13)
}

ParameterID ::= ENUMERATED {
allparameters (0),
numberofantennas (1),
activechannelnumber (2),
activechannelbandwith (3),
activechannelcenterfrequency (4),
activechanneltxpower (5),
activechanneldatarate (6),
camgenerationrate (7),
camechoingmode (8),
demngenerationmode (9),
demnechoingmode (10),
clientappip (11),
clientappcport (12),
serverrequesttimeout (13)
}

Opaque ::= SEQUENCE {
datatype DataType,
data BIT STRING
}

Data ::= CHOICE {
opaque Opaque,
cam CAMp2aMessage
}

SendRequestMessage ::= SEQUENCE {
data Data
}

SetRequestMessage ::= SEQUENCE {
parameters Parameter
}

StatusRequestMessage ::= SEQUENCE {
parameterid INTEGER
}

ResponseRequestMessage ::= SEQUENCE {
error Error
}

ReceiveRequestMessage ::= SEQUENCE {
data Data
}

NotificationRequestMessage ::= SEQUENCE {
error Error,
parameters Parameter
}
```

```
Payload ::= CHOICE {
sendrequest SendRequestMessage,
receiverequest ReceiveRequestMessage,
setrequest SetRequestMessage,
statusrequest StatusRequestMessage,
resposerequest ResponseRequestMessage,
notificationrequest NotificationRequestMessage
}

MessageP2a ::= SEQUENCE {
messageheader MessageHeader,
payload Payload

}
```

5 PROTOCOL EVALUATION

The introduced overhead and overall message size were measured through the implementation of a Java application. One advantage of using ASN.1 is the multitude of tools available that generate the code automatically, given the message definitions.

So, the application developed uses the ASN.1 for the message definitions, which are then encoded using the Basic Encoding Rules (BER) (Mitra, 1994). These define how information can be encoded using binary. BER uses a Type Length Value (TLV) structure, as shown in Figure 2. The three components of the structure can vary in size. The first (type) has, typically, a fixed size of 1 byte. The number of bytes occupied by the size information depends on the size of the next component (value). A single byte is enough if the information is smaller than 127 bytes. However, if the value has 128 bytes or more, the length component will need more than 1 byte. The last component is the actual data of the element.

Type
Length
Value

Figure 2: BER TLV structure.

The message types previously defined can be divided into two types, depending on their format. *SendRequest* and *ReceiveRequest*, may have different payloads with different sizes depending on the data size or if they are *Opaque* or *CAM*. If the data type is *CAM*, the data size is also fixed.

The rest of the messages, *NotificationRequests*, *SetRequest*, *StatusRequest*, *NotificationRequest*, and *ResponseRequest* do not have payloads with different sizes but fixed parameters and error messages.

In the first group of messages, their message size and, more specifically, the overhead introduced was evaluated and is shown in Table 4. In Table 5, the message size for the other message types is shown. To be noticed that all the messages have an **ID**. So, the total overhead of the messages can vary depending on this value. This table also includes *SendRequests* and *ReceiveRequests* when used with CAMs as the payload because, in this case, they also have fixed sizes.

Table 4: Message size comparison (Bytes).

Payload Size	Encoding Size	Overhead
1	25	24
10	34	24
100	124	24
200	230	30
300	336	36
1000	1036	36

Table 5: Message size comparison (Bytes).

Message Type	Encoding Size
SendRequest CAM	42
ReceiveRequest CAM	42
ResponseRequest	19
SetRequest	64
StatusRequest	17
NotificationRequest	67

Table 4 presents the overhead introduced in the messages that may have different size payloads (*SendRequest*, *ReceiveRequest*). As both message types have the same format, their size and overhead are equal. The columns indicate the original size of the data payload, the size after the encoding process, and the total overhead. All the mentioned message types introduce the same overhead, so only one table is presented for all of them. The overhead introduced depends on the payload size being 24 bytes for payloads smaller than 200 bytes, 30 bytes for messages smaller than 200 bytes, and 36 bytes for messages bigger than 300 bytes. Thus, the introduced overhead is very small, and it is almost negligible for messages bigger than 100 bytes. The overhead increases with the size of the message due to the TLV structure of the BER encoding. As previously stated, the length component of the TLV is variable and increases with the size of the data encoded.

The size of the other messages can be seen in Table 5. This table shows the message size after encoding for each message type that does not have a variable payload size.

Any of the defined messages with a fixed size is smaller than 70 bytes. The biggest one is the *NotificationRequest* and the smallest *StatusRequest*. So, it seems to have small enough sizes for in-vehicle communications, either over wireless or wired mediums.

The evaluation of the protocol only considered the overhead introduced by the multiple message types. Encoding speed or delay introduced are metrics that heavily rely on external factors. The encoding speed depends on the supporting device characteristics or the coding of the application. The delay introduced can be measured in two different places. The first is during the message encoding. This metric suffers the same problem of the encoding speed. The other is the delay introduced in the message exchange. As the overhead introduced is very small and ideally transmitted over high bandwidth mediums, the delay introduced in the connection is negligible.

6 CONCLUSIONS

This paper presents an architecture for an agnostic middleware and its corresponding API, as well as a protocol specification and implementation. The presented protocol allows third-party applications to easily take advantage of an agnostic architecture. It allows applications to easily access the vehicle's data sources without any knowledge of the OBU internals.

The API's messages are defined using the ASN.1 notation over the UDP protocol. ASN.1 is a widely used and accepted format for describing data transmitted over communication protocols and is independent of the data's implementation language and physical representation, permitting its easy adoption.

The protocol was evaluated in terms of overhead introduced. Evaluating the encoding speed and communication time was not performed because it depends on external factors. The overhead introduced is minimal, with only 24 bytes in messages of 100 bytes and 36 for messages of more than 300 bytes.

The current implementation does not have any security considerations. It is an in-vehicle protocol communication deployed in the same local area network, which is considered a secure environment. Nonetheless, it is to be investigated the impact and need for security mechanisms.

The API and protocol are functional and allow access to the configuration parameters and services. It was the basis for the implementation of a platooning application tested with success in the real world. Currently, the OPENC2X (Laux et al., 2016; Klingler et al., 2017) platform is being extended to support the developed API. Nonetheless, it is still evolving, with new functionalities being implemented.

ACKNOWLEDGEMENTS

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

REFERENCES

Costa, R. (2013). VADM - A common API for agnostic application development within VANETs. In *Proceedings Elmar - International Symposium Electronics in Marine*, pages 365–368.

Dias, B. et al. (2018). Agnostic and Modular Architecture for the Development of Cooperative ITS Applications. *Journal of Communications Software and Systems*, 14(3):218–227.

Google (2022). Protocol Buffers. Online, https://developers.google.com/protocol-buffers (accessed on 2022-04-20).

Harrington, D. et al. (2002). An architecture for describing simple network management protocol (snmp) management frameworks.

ITU-T (2022). ASN.1. Online, https://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx (accessed on 2022-04-20).

Jawhar, I. et al. (2013). An overview of inter-vehicular communication systems, protocols and middleware. *Journal of Networks*, 8(12):2749–2761.

Klingler, F. et al. (2017). Poster: Field Testing Vehicular Networks Using OpenC2X. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, page 178, New York, NY, USA. Association for Computing Machinery.

Laux, S. et al. (2016). Demo: OpenC2X — An open source experimental and prototyping platform supporting ETSI ITS-G5. In *2016 IEEE Vehicular Networking Conference (VNC)*, pages 1–2.

Mitra, N. (1994). Efficient Encoding Rules for ASN.1-Based Protocols. *AT&T Technical Journal*, 73(3):80–93.

Nour, S. et al. (2011). Middleware for Data Sensing and Processing in VANETs. In *2011 International Conference on Emerging Intelligent Data and Web Technologies*, pages 42–48. IEEE.

Silva, F. A. et al. (2013). ConProVA: A smart context provisioning middleware for VANET applications. In *IEEE Vehicular Technology Conference*.

Sousa, S., Santos, A., Costa, A., Dias, B., Ribeiro, B., Gonçalves, F., Macedo, J., Nicolau, M. J., and Gama, Ó. (2017). A New Approach on Communications Architectures for Intelligent Transportation Systems. *Procedia Computer Science*, 110:320–327.

WG16, I. T. C. . (2014). ISO 21217: 2014: Intelligent transport systems-Communications access for land mobiles (CALM)-Architecture.

Willke, T. et al. (2009). A survey of inter-vehicle communication protocols and their applications. *IEEE Communications Surveys and Tutorials*, 11(2):3–20.